分散メモリ型並列計算機の自動並列化コンパイラ
—Inspector/Executorアルゴリズムの高速化—

窪田 昌史[†]，三吉 郁夫，大野 和彦，森 眞一郎，中島 浩，富田 眞治
京都大学 工学部 († :現在 日本IBM(株))

本稿では，分散メモリ型の並列計算機に対するSPMDコード生成技法について述べる．インデックス配列による間接アクセスが存在するループを並列化すると，不規則なアクセスパターンを生ずる．従来insepctorとexecutorというコードを生成する手法が提案されてきたが，inspectorにおいて全対全のプロセッサ間通信が必要であり，適用できるコードの範囲にも制限がある．これらの問題を解決するために，逆インデックス法と全検査法という2つのinspectorのアルゴリズムを提案する．さらに，それらの手法の有効性を高並列計算機AP1000上で評価した．

In this paper, we focus on parallelizing compiler techniques which generate SPMD codes for distributed memory computers. Parallelization of loops with indirect accesses with index arrays causes irregular access patterns. For such codes, a technique to generate inspector/ executor code has been proposed. In this technique, however, the inspector must perform all-to-all global communications. Furthermore, codes, to which this method is applicable, are restricted. In order to resolve these drawbacks, we propose two inspector algorithms, inverse index method and exhaust inspection method. We evaluated the effectiveness of these methods on the highly parallel computer AP1000.

1 はじめに

現在我々は，科学技術計算を対象として，分散メモリ型並列計算機へのプログラムの自動並列化，並列化支援の研究を行っている．

科学技術計算では，実行時間の大部分がループ内での行列に対する操作や演算に費やされる．ループ部分は，細粒度で規則性をもつ並列性を内在しているため，全てのプロセッサが同一のプログラムを実行し，ループ部分をSIMD的に各プロセッサに割り当てるSPMD(Single Program Multiple Data Stream)モデルによるプログラムの並列化が有効である．SPMDのモデルに基づく並列言語や並列化コンパイラの研究は数多く行われている([1]など)．

本稿では，コード生成部におけるコンパイル技法について議論する．データ分割はディレクティブなどで指定されているFORTRANなどの手続き型言語で記述された逐次プログラムを対象とする．また，プログラム中のデータ依存関係による並列性の抽出などは，ソースプログラムの解析によって行われていることを前提とする．

本稿では，2章で，我々が開発中の並列化コンパイラの基本方針と，DOALL型ループ内に起こる不規則アクセス，およびこのようなループに対するInspector/Executorというコード生成戦略について述べる．3章では，この戦略のinspector部分のアルゴリズムに的を絞り，逆インデックス法と全検査法という新たなアルゴリズムを提案する．4章では，行列のLU分解や疎行列とベクトルの積へのアルゴリズムの適用について述べ，その有効性を評価する．最後に5章で総括を行い，今後の課題にも触れる．

2 並列化コードの生成

2.1 基本戦略

並列化コードの生成は，

- データ分割に基づくプロセッサへのデータ割付け
- 割り付けられたデータに基づく，各命令のプロセッサへの割付け
- データ参照に伴うメッセージ通信のコードの挿入

というフェーズに分かれる．以下，これらの基本戦略を説明する．

Automatic Parallelizing Compiler for Distributed Memory Parallel Computers —New Algorithms to Improve the Performance of the Insepctor/Executor—
Atsushi KUBOTA[†], Ikuo MIYOSHI, Kazuhiko OHNO, Shin-ichiro MORI, Hiroshi NAKASHIMA, Shinji TOMITA
Faculty of Engineering, Kyoto University
†current affiliation: IBM Japan, Ltd.

データ分割　配列の分割法としては，BLOCK, CYCLIC 分割が有効であることが経験的に知られている．本コンパイラにおいても，当面はこの２種類の分割を対象とする．

コンパイラへのデータ分割の指定は，配列に対するディレクティブによって行う．このような規則的な分割法を採用すると，データのプロセッサへの割り付けはコンパイル時に行うことができる．

命令のプロセッサへの割り付け　割り付けられたデータに対応するコード生成には，Owner Computes Rule[2] に基づく手法を採用する．これは，分割された配列データを保持するプロセッサ (*owner*) に，そのデータをアクセスする命令を割り付けるという戦略である．代入文では，原則として左辺のデータを所有するプロセッサにその代入文が割り付けられる．

通信コードの挿入　アクセスすべきデータがプロセッサ内にあるか，他のプロセッサとのメッセージ通信によって得られるかの判定は原則としてコンパイル時に行われる．その結果，必要であればメッセージ通信のコードが生成される．

2.2　不規則アクセスを伴うループ

配列の添字内の整数型配列によるインデックス参照が存在する場合，アクセスすべきデータの位置が実行時に決定されるため，コンパイル時にプロセッサ間のメッセージ通信の要／不要を決定できない．このようなデータアクセスを本稿では，不規則アクセス (irregular accesses) と呼ぶ．

部分ピボッティング付きの LU 分解[1]，疎行列の処理など，非常によく使われるプログラムの中にも不規則アクセスを含むものが多い．図 1(a) の例は，不規則アクセスを生ずる DOALL 型ループである．

Owner Computes Rule に基づいて，`a(i)` を保持するプロセッサが `i` 番目のイタレーションを実行することはコンパイル時に決定できるが，`b(index(i))` を保持するプロセッサは実行時にならなければ確定しない．そのため，`b(index(i))` を送信するプロセッサは実行時に決定される．

ゆえに，`b(index(i))` のアクセスのたびに `index(i)` の値を調べ，配列 `b(index(i))` の要素を保持するプロセッサへリクエストのメッセージを送信し，これを受信したプロセッサが `b(index(i))` の値を送り返すことが必要である．

```
do i=1,N
  a(i) = b(index(i))
enddo
```

(a) 右辺のみインデックス配列

```
do i=1,N
  a(index1(i)) = b(index2(i))
enddo
```

(b) 両辺にインデックス配列

図 1: DO ALL 型ループ

```
do $\vec{k}$=$\vec{l}$,$\vec{m}$,$\vec{s}$
  $X(g(\vec{k}))=Y_1(h_1(\vec{k}))\circ Y_2(h_2(\vec{k}))\cdots Y_n(h_n(\vec{k}))$
enddo
```

$\circ$は，+などの演算を表す．ベクトル表記は多重ループを表す．

図 2: DO ALL 型ループの一般形

2.3　Inspector/Executor 戦略

実行時のデータアクセスの度に送受信プロセッサを確定するとオーバーヘッドが大きい．そのため，図 2に示すような DOALL 型ループに対して，先にプロセッサ間通信を伴うデータ参照を検査する **inspector** を実行し，引き続いて演算そのものを行う **executor** を実行するというコード生成法が提案されている [3]．本稿ではこのコード生成戦略を **Inspector/Executor** 戦略と呼ぶ．以下にプロセッサ p におけるコードの概要を示す．なお，この中でプロセッサ qは $p \neq q$となる全てのプロセッサである．

1. inspector では，以下のリストが作成される．
   (a) プロセッサ間通信のためのデータ参照関係を示すリストの作成
      - p が参照し，かつ，qが所有している配列データの変数名とその添字のリスト **recv_list(p,q)**
      - p が所有しており，かつ，qが参照する配列データの変数名とその添字のリスト **send_list(p,q)**
   (b) executor でのイタレーションの実行順序を決めるためのリストの作成

---

[1] 以下，LU 分解という場合，特に断らない限り部分ピボッティング付きのものを指す．

  - p において，ローカルに割り付けられているデータアクセスのみで実行されるイタレーションのリスト **local_iter(p)**
  - p 以外のプロセッサからのデータもアクセスする必要があるイタレーションのリスト **nonlocal_iter(p)**

2. executor では，データの送受信およびループの実行が以下の順で行われる．
   (a) *send_list(p,q)* に基づいて，qへデータを送信する．
   (b) イタレーション *local_iter(p)* のループを実行する．
   (c) *recv_list(p,q)* のデータを qから受信する．
   (d) 受信したデータを参照するイタレーション *nonlocal_iter(p)* を実行する．

executor では，送信すべきデータを先送りし，通信が行われている間にデータ・アクセスに通信を必要としないイタレーション *local_iter*(p) を実行する．ついで必要とするデータを受信し，残りのイタレーション *nonlocal_iter*(p) を実行する．このように，executor において，あるプロセッサ p から q への通信を一度にまとめることで通信オーバヘッドを低減するとともに，ループのイタレーションの実行順序を変更することで通信のレイテンシの隠蔽を図っている．inspector はこれらの実現のための前処理という役割を持つ．

3 Inspector アルゴリズムの高速化

本章では，Inspector/Executor 戦略に対して逆インデックス法と全検査法という新たなアルゴリズムを提案する．これらのアルゴリズムと [3, 4] で提案されたアルゴリズム（以下，従来法と呼ぶ）との大きな違いは，inspector での *send_list* の作成法にある．

従来法では，$send\_list(p,q) = recv\_list(q,p)$ であることを利用し，*recv_list* をプロセッサ間で通信することにより，*send_list* を求める．これに対し，我々の提案する逆インデックス法では，インデックス配列の逆写像である逆インデックス配列を求め，通信を行わずに *send_list* を求める．また，全検査法では全てのインデックス配列を参照することで，逆インデックスを用いずに，かつ，通信を行わずに *send_list* を求める．

```
do $\vec{j} \in$ {プロセッサ p で実行するイタレーションの集合}
  local_flag = true;
  /* $recv\_list_{Y_i}(p, \mathtt{src})$ を求める */
  do each $Y_i$
    src = $Y_i(h_i(\vec{j}))$ を所有するプロセッサ番号;
    if (src != p) then
      local_flag = false;
      append $h_i(\vec{j})$ to $recv\_list_{Y_i}(p, \mathtt{src})$;
  endif enddo
  /* $local/nonlocal\_iter$ を求める */
  if (local_flag == true) then
      append $\vec{j}$ to $local\_iter(p)$;
  else
      append $\vec{j}$ to $nonlocal\_iter(p)$;
endif enddo
```

図 3: 従来法，逆インデックス法の *recv_list*, *local/nonlocal_iter* を求めるアルゴリズム

3.1 従来法の inspector のアルゴリズム

1. 自分のプロセッサ内で必要とするデータのリスト *recv_list*，executor のイタレーションの順番を変更するためのリスト *local/nonlocal_iter* を作成する（図 3）．
2. *recv_list* を他のプロセッサへ送信し，逆に，他のプロセッサから *recv_list* を受信する．受信したリストは *send_list* に相当する．

例えば，図 1(a) では，`a(i)` の分割に従って，あるプロセッサ（p とする）で実行すべきイタレーションも規則的に決定できる．そのイタレーション番号を右辺のデータのインデックス配列 `index(i)` に代入すれば，`b(index(i))` を所有しているプロセッサ（q とする）が決定できる．ゆえに，q から p への通信が必要であることがわかり，`index(i)` の値を *recv_list(p,q)* に加えることになる．また，*local/nonlocal_iter* は，*recv_list* を作成する過程で，あるイタレーションの実行にメッセージ通信を必要とするかどうかを判定しながら決定される．

このアルゴリズムの問題点は，*send_list* を求めるために全対全通信を行っており，通信のオーバーヘッドが大きくなることである．

また，図 1(b) のように左辺にもインデックス配列が存在する場合，そのインデックス配列の逆写像にあたる逆インデックス配列が求められなければ，あるプロセッサにおいて，何番目のイタレーションを実行すべきかを求めることができない．[3] では，このように左辺にインデックス配列が現れる場合を考察の対象から外している．従って，左辺

```
do each $Y_i$
  do $\vec{j} \in deref_{Y_i}(p)$
    dest = $X(g(\vec{j}))$ を所有するプロセッサ番号;
    if (dest != p) then
      append $h_i(\vec{j})$ to $send\_list_{Y_i}(p, \mathrm{dest})$;
endif enddo enddo

$\mathrm{deref}_{Y_i}(\mathrm{p})$ = {右辺の配列データ $\mathrm{Y}_i$ のうちプロセッサ p
                が所有するデータを参照するイタレーションの集合}
```

図 4: 逆インデックス法の *send_list* を求めるアルゴリズム

にインデックス配列が現れる LU 分解などのコードには適用不可能である．

3.2 逆インデックス法によるアルゴリズム

前節で指摘した，*send_list* の作成に通信を行うことと，代入文の左辺にインデックス配列が存在する場合に適用できないという従来法の問題点を，逆インデックス配列を積極的に活用することで解決するアルゴリズムを提案する．以下に，逆インデックス法のアルゴリズムを示す．

1. 右辺の配列の逆インデックス配列を求めて，他のプロセッサが必要としているデータのリスト *send_list* を作成する（図 4）．
2. 自分のプロセッサ内で必要とするデータのリスト *recv_list*，executor のイタレーションの順番を変更するためのリスト *local/nonlocal_iter* を作成する．このフェーズは，従来法の 1. と同一である（図 3）．

まず，図 1(a) において，*send_list* が通信を行わずに求められることを示す．あるプロセッサ p に保持されている配列データ `b(i)` の添字番号は規則的であるため容易に求められる．それらローカルに保持されている右辺の配列データを参照するイタレーションは，`index(i)` の逆インデックスによって求められる．参照するイタレーションがわかれば，それを実行するプロセッサ（プロセッサ q とする）は容易に求められるので，プロセッサ p から q へ通信が必要なことがわかり，`index(i)` の値が *send_list(p,q)* に加えられる．

また，両辺にインデックス配列が存在する図 1(b) では，ローカルに保持されている左辺のデータ a の要素について，インデックス配列 `index1(i)` の逆インデックスをとることで，あるプロセッサ p で実行すべきイタレーション番号が求められる．このイタレーション番号により右辺のデータのインデックス配列 `index(i)` を索引すれば，そのイタレーションで参照されるデータ `b(index2(i))` を保持するプロセッサ q が求まるので，`index2(i)` を *recv_list(p,q)* に加えればよい．

```
do $\vec{j} \in$ {逐次ループの全てのイタレーション}
  local_flag = true;
  dest = $X(g(\vec{j}))$ を所有するプロセッサ番号;
  do each rhs $Y_i$
    src = $Y_i(h_i(\vec{j}))$ を所有するプロセッサ番号;
    /* $send\_list_{Y_i}(p, \mathrm{dest})$ を作成する */
    if (src == p) then
      if (dest != p) then
        append $h_i(\vec{j})$ to $send\_list_{Y_i}(p, \mathrm{dest})$;
      endif
    /* $recv\_list_{Y_i}(p, \mathrm{src})$ を作成する */
    else  /* (src != p) */
      local_flag = false;
      if (dest == p) then
        append $h_i(\vec{j})$ to $recv\_list_{Y_i}(p, \mathrm{src})$;
  endif endif enddo
  /* $local/nonlocal\_iter$ を求める */
  if (dest == p) then  /*
    if (local_flag == true) then
      append $\vec{j}$ to $local\_iter(p)$;
    else
      append $\vec{j}$ to $nonlocal\_iter(p)$;
endif endif enddo
```

図 5: 全検査法の inspector アルゴリズム

逆インデックス配列の指定 一般にインデックス配列の逆インデックス配列を求めることは困難である．しかし，LU 分解におけるピボットのように，インデックス配列が 1 対 1 の置換群 (permutation) となる場合，この性質がコンパイラに与えられていれば，実行時に逆インデックス配列を生成することが可能である．

そこで，

```
index permutation pivot(1:N)
```

のようにプログラマがインデックス配列の性質をディレクティブで指定し，そのインデックス配列をプログラマが操作する場合は，`swap(pivot,i,j)` や `shuffle(pivot,k)` といったインデックス値の交換や $k-$シャフル置換などのライブラリ関数に限るものとする．コンパイラはこのディレクティブに従って逆インデックス配列を求めるコードを生成する．

3.3 全検査法によるアルゴリズム

このアルゴリズムでは，全てのインデックス配列の全ての要素を各プロセッサが保持し，その全てを検査する（図5）．つまり，図1(b)なら，`i=1,N`の全てのイタレーション番号により両辺のインデックス配列`index1,index2`を索引する．その結果，あるプロセッサpが`a(index1(i))`または`b(index2(i))`を保持していることがわかれば，プロセッサpがそのイタレーションに関わっている（実行，通信）ことになる．そのようなイタレーションについては，データの参照関係を求め，適切なリストの操作を行う．

このアルゴリズムでは，各プロセッサがインデックス配列のコピーを保持するだけで適用できる．そのため，コードの逆インデックス配列の生成可能性にかかわらず適用できるという特徴を持つ．

3.4　アルゴリズムの比較

前節の3種のアルゴリズムを，通信の有無，必要となるインデックス配列，inspectorのループの実行時間の観点から比較する（表1）．

表1: inspectorのアルゴリズムの比較

アルゴリズム		従来法	逆index法	全検査法
通信回数		$P-1$	不要	不要
左辺にindex配列が存在しない場合				
右辺	index	local	local	all
	逆index	不要	local	不要
左辺にindex配列が存在する場合				
左辺	index	適用不可能	all	all
	逆index		local	不要
右辺	index		all	all
	逆index		local	不要

(allは，インデックス配列の全てを所有する必要があり，localは，自プロセッサにローカルなデータに対応する分だけを分割して所有していれば良いことを示す．)

inspector内での通信　逆インデックス法，全検査法では通信が不要である．それに対して，従来法では全対全通信が必要なので，送受信ともに$P-1$回必要である．

アルゴリズムの適用条件　inspectorのアルゴリズムを適用できるためのインデックスの条件は，代入文の左辺にインデックス配列が存在するか否かで異なる．

従来法では，実行すべきイタレーション番号を得るために，左辺の添字式の逆関数を求める必要がある．そのため，左辺にインデックス配列が存在すると適用不可能となる．

逆インデックス法では，左辺，右辺の項の添字式の逆関数と，これらのイタレーションに対する添字が，全て求められる場合に適用可能である．

全検査法では，全てのインデックス配列を必要とするが，逆インデックス関数は全く不要であるため適用範囲が広い．

inspectorでのループの回数　プロセッサ数Pと，配列データのサイズNに大きく依存する．右辺の項数をr，$send/recv_list$を求めるループの1イタレーションの実行時間の平均をT_L，$local/nonlocal_iter$を求める時間の合計をT_I，従来法のinspectorの1回のメッセージの送受信にかかる平均時間をT_Mとすると，inspectorの実行時間は以下の式で近似できる．

$$\text{従来法} = \frac{rNT_L}{P} + T_M(P-1) + T_I \qquad (1)$$

$$\text{逆インデックス法} = \frac{2rNT_L}{P} + T_I \qquad (2)$$

$$\text{全検査法} = 2rNT_L + T_I \qquad (3)$$

これらの式から，inspectorの実行時間について定性的に考察する．従来法では，プロセッサ数に比例する項と，反比例する項が存在するため，プロセッサ数を増加すると，inspectorの実行時間は一度減少した後，$P = \sqrt{rNT_L/T_M}$で最小となり，さらにプロセッサ数を増加すると実行時間が増加してしまう．それに対して，逆インデックス法では，プロセッサ数の増加とともにinspectorの実行時間が単調に減少する．また，全検査法ではプロセッサ数にかかわらず実行時間は一定になる．

4　Inspector/Executor戦略の有効性の評価

本章では，LU分解，不規則疎行列とベクトルの積の例題について，3章で述べたアルゴリズムを適用したハンドコンパイルによるプログラムを高並列計算機AP1000[5]上で実行し，時間を計測して得られた結果をもとに，アルゴリズムの有効性を検証する．

4.1　部分ピボッティング付きLU分解

LU分解のコードを図6に示す．これを，Inspector/Executor戦略を適用しなかった場合（以下，

```
 decomposition d(N,N)
 align a(i,j)  with d(i,j)
 align pivot(i)  with d(i,:)
 distribute d(CYCLIC)(CYCLIC)
do k=1,N
  /* ピボットの選択，交換 */

  /* 分解 */
  do i=k+1,N
    do j=k+1,N
      a(pivot(i),j) = a(pivot(i),j)
        - a(pivot(i),k) * a(pivot(k),j)
    enddo
  enddo
enddo
```

図 6: 部分ピボッティング付き LU 分解のコード

単純法と呼ぶ）と Inspector/Executor 戦略の逆インデックス法と全検査法を適用した場合で実行し，その実行時間を計測した．

4.1.1 逆インデックス配列

`pivot` は，permutation であるため逆インデックスの生成が可能となる．

`pivot` の逆インデックスの更新は，`pivot` が更新されるときに，各プロセッサが対応する逆インデックス配列の要素を保持しているかを検査して行う．逆インデックス配列を生成するために新たなプロセッサ間通信は生じない．

4.1.2 実行結果

AP1000 において，サイズ $N = 64, 128, 256$ の $N \times N$行列の LU 分解の実行時間を測定した．

行列 `A` は，行方向，列方向の両方向について CYCLIC 分割して並列化を行った．図7に $N = 256$ の実行時間を示す．

なお，従来法は適用不可能である．

4.1.3 考察

図 7から，Inspector/Executor 戦略を適用した場合，大幅な高速化が行われることがわかる．逆インデックス法を使用して Inspector/Executor 戦略を適用した場合の実行時間は，単純法の最高 42 倍（プロセッサ数 512）まで高速化された．また，全検査法では，同じ条件で実行時間が最高 11 倍まで高速化された．このような良好な結果を得たのは，データアクセスの度に起こるブロードキャストの回数を，大幅に削減したためである．

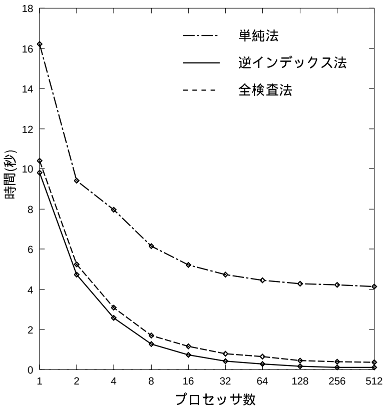


図 7: LU 分解の実行時間 (256×256)

逆インデックス法は全検査法に比べ，プロセッサ数 512 で実行時間が 3.7 倍 高速になっている．これは，式 (2) に示すように，逆インデックス法ではプロセッサ数が増加するにつれて inspector のコストが小さくなるためである．この差は，プロセッサ数の増加につれて大きくなるため，大規模な並列システムにおいて，逆インデックス法が有効であるといえる．

4.2 不規則疎行列とベクトルの積

有限要素法などの問題で必要となる，不規則疎行列とベクトルとの積のプログラム (以下 SMVM:Sparse Matrix-Vector Multiply と略す）を用いて評価を行う．アルゴリズムは，Insepctor/Executor 戦略を用いない単純法と，3章で述べた 3 種類の inspector アルゴリズム (従来法，逆インデックス法，全検査法）を用いる．これらのアルゴリズムを適用したコードをハンドコンパイルによって記述し，その評価を行う．図 8に逐次の SMVM のコードを示す．

4.2.1 逆インデックス配列

不規則疎行列 `A` は非零要素が不規則な位置に存在し，かつ，非零要素の個数が行列のサイズ Nに対して非常に少ないものである．そこで，記憶効率の向上のため図 9のような配列に格納されることが多い．

1. 第 `i` 行の `(i,i+1)` 要素から `(i,N)` 要素のうち，非零要素の個数を `nzu(i)` とする．

```
decomposition d(N)
align nzu(i),iu(i,*),au(i,*) with d(i)
align nzl(i),il(i,*),al(i,*) with d(i)
align ad(i),b(i),x(i) with d(i)
distribute d(BLOCK)

do i=1,N
  x(i) = 0.0;
  do j=1,nzl(i)
    x(i) = x(i) + al(i,j) * b(il(i,j))
  enddo;
  x(i) = x(i) + ad(i) * b(i)
  do j=1,nzu(i)
    x(i) = x(i) + au(i,j) * b(iu(i,j))
  enddo
enddo
```

図 8: 不規則疎行列 A とベクトル b の積 x を求めるプログラム

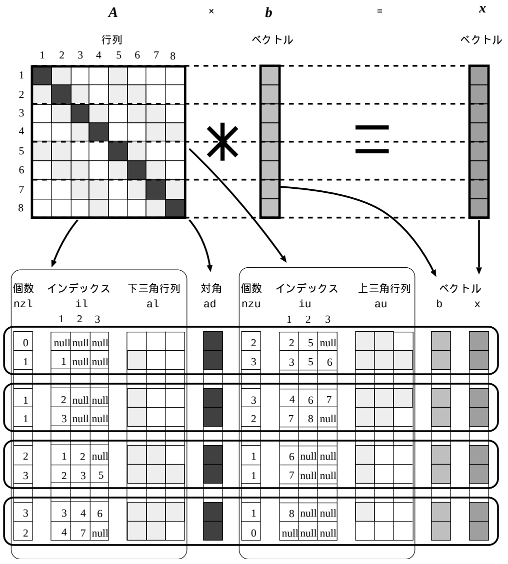
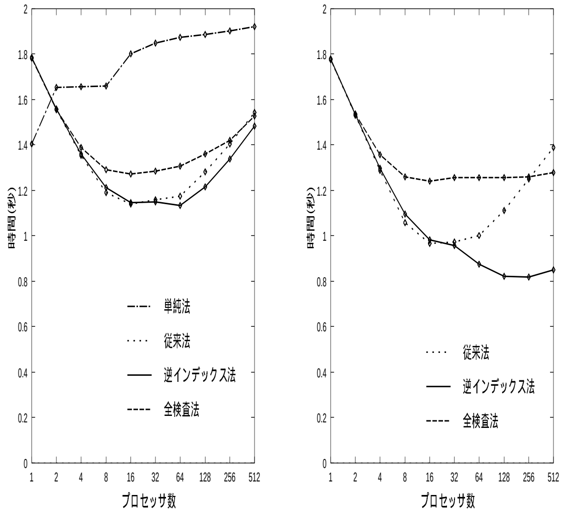


図 9: 不規則疎行列とベクトルの積

2. 第 `i` 行において，`(i,j)` 要素 （但し $i+1 \le j \le N$）中の非零要素を列番号の昇順で取り出したとき，第 `j` 番目の要素の存在する列番号を `iu(i,j)` とする．
3. `iu(i,j)` に対応する非零要素を `au(i,j)` とする．
4. 第 `i` 行の対角要素を `ad(i)` とする．

1. から 3. で表現されるのは上三角行列の部分であり，下三角行列についても第 `i` 行の非零要素の個数，要素の存在する列番号，非零要素の値をそれぞれ，`nzl(i)`，`il(i,j)`，`al(i,j)` に格納する．



(a) 全体 (b)inspector

図 10: SMVM の実行時間

特に，行列 `A` が対称行列の場合は，`il` から `iu` の逆インデックスが，`iu` から `il` の逆インデックスが，それぞれ求まる．したがって，逆インデックス法を適用することができる．

また，非対称行列の場合も，行方向についてのインデックス配列と列方向のインデックス配列の両方を用意しておけば，互いの逆インデックス配列を求めることができる．

4.2.2 実行結果

AP1000 において，SMVM の実行時間を測定した．プロセッサ数は LU 分解と同じく P=1 から 512，行列のサイズは，N=64, 256, 1K, 4K, 16K, 32K とし，1 行あたりの非零要素数は対角要素を除いて最大 6 個である．実行時間の測定は，SMVM のサブルーチン全体と，その中の inspector 部のみの 2 種類で行った．行列 A やベクトル b, x は図9に示すように，行方向に BLOCK 分割した．図10に行列のサイズが $N = 1K$ の場合の SMVM 全体の実行時間と inspector のみの実行時間を示す．

4.2.3 考察

単純法との比較 図 10(a) から，Inspector/Executor 戦略を用いた全てのアルゴリズムで実行の高速化が達成されていることがわかる．これが最高となるのは，プロセッサ数 $P = 64$ 台，逆インデックス法で実行した場合で，実行時間は 1.65 倍まで高速化された．同じく，$P = 64$ で，従来法の

inspector では最高 1.59 倍，全検査法では 1.43 倍の高速化が達成された．

従来法の inspector との比較　SMVM の逆インデックス法，全検査法による inspector のアルゴリズムが従来法のアルゴリズムに対してどの程度高速化されているかを算出すると，逆インデックス法では，プロセッサ数 $P = 512$ で 1.63 倍，全検査法では，同じく $P = 512$ で 1.09 倍まで高速化されている．

図10(b) の従来法では，プロセッサ台数の増加とともに，inspector の実行時間は一度減少し，プロセッサ数 16 の場合を最小として再び増加する．これに対して逆インデックス法ではプロセッサ数の増加に伴い，inspector の実行時間が減少しており，通信を不要にした効果が現れている．また，全検査法では全てのインデックスを検査するため，プロセッサ数が増加しても inspector の実行時間はほぼ一定となっている．これら inspector の実行時間は，3.4で述べた理論式に良く合致していると言える．

5　おわりに

本稿では，不規則なアクセスを伴うループの並列化において，すでに提案されている Inspetor/Executor 戦略の inspector 部分を高速化し，かつ，適用範囲を拡張するアルゴリズム（逆インデックス法ならびに全検査法）を提案した．本稿で提案したアルゴリズムの特徴は，プロセッサ間通信を行わずに inspector 部分が実行可能な点にある．

逆インデックス法では，インデックス配列の逆写像である逆インデックス配列をもちいることで従来法で必要であった通信を排除する．全検査法では，全てのインデックス配列を検索することで，逆インデックスを用いることなく通信を排除する．

これにより，全検査法ではアルゴリズムの適用範囲の制限をなくすことが可能となり，逆インデックス法でも，従来適用できなかったピボッティング付き LU 分解等の多くの問題に適用可能となった．

また，並列計算機上での評価結果より，逆インデックス法が従来法よりも高速であることが示された．さらに，プロセッサ台数が増加すると全検査法も従来法より高速であることも示された．

今後の課題として，*send_list*, *recv_list* などのリスト操作の高速化，より一般的な問題にたいする逆インデックス配列の生成可能性の評価，ならびに，インデックス配列の性質のコンパイラへの指定法の検討等を考えている．

謝辞

テストプログラムの実行にあたり，高並列計算機 AP1000 の実行環境を御提供いただきました（株）富士通研究所，プログラム開発環境を御支援いただきました横河・ヒューレット・パッカード（株）に感謝の意を表します．

また，日頃より有益な御意見をいただく富田研究室の諸氏に感謝致します．

なお，本研究の一部は文部省科学研究費補助金（重点領域研究 (1) 課題番号 04235103「超並列ハードウェア・アーキテクチャの研究」）による．

参考文献

[1] Joel Saltz and Piyush Mehrotra, editors. *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*. Elsevier Science Publishers B. V., 1992.

[2] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, pp. 86–100, Albuquerque, NM, November 1991.

[3] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel and Distributed Syst.*, Vol. 2, No. 4, pp. 440–451, October 1991.

[4] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in Fortran D. In Saltz and Mehrotra [1], pp. 139–176.

[5] 清水俊幸, 堀江健志, 石畑宏明. 高速メッセージハンドリング機能 –AP1000 における実現–. 並列処理シンポジウム JSPP '92, pp. 195–202, June 1992.